CEWES MSRC/PET TR/98-32

Towards a Fortran 90 Interface to the POSIX Threads Library

by

Clay P. Breshears
Henry A. Gabb
S. W. Bova

05h00898

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC 94-96-C0002
Nichols Research Corporation

Towards a Fortran 90 Interface to the POSIX Threads Library

Clay P. Breshears*[1]
CEWES MSRC
Vicksburg, MS 39180
clay@turing.wes.hpc.mil

Henry A. Gabb
CEWES MSRC
Vicksburg, MS 39180
gabb@ibm.wes.hpc.mil

S.W. Bova[2]
CEWES MSRC
Vicksburg, MS 39180
bova@gonzo.wes.hpc.mil

Abstract

Pthreads is a POSIX standard established to control the spawning, execution and termination of multiple threads within a single process. Because of a much lower overhead, use of Pthreads is an attractive approach. Under this programming paradigm on a shared-memory system, threads execute within a single address space, although multiple processors may be employed to execute the various threads. The disadvantage of Pthreads with respect to high performance computing is that there is no Fortran interface defined as part of the POSIX standard. We present our current progress in defining and implementing a full set of Fortran 90 bindings for the POSIX threads functions. Also presented are many of the design decisions made and lessons learned while striving to keep the bindings as portable as possible. A particle dynamics code is being used as a test platform to evaluate the feasibility, efficiency and scalability of such a parallel execution model. Preliminary results for the particle dynamics test code indicate that the per-processor performance may be slightly less than with compiler directives, but that the scalability is superior. In addition, our investigation of combining Pthreads with MPI is described. This could be a useful programming model for SMP clusters.

1. Introduction

In dynamics simulations, the through-space interactions between particles, which must be calculated every time step, consume the bulk of computational time. These calculations typically occur in a single, large loop containing many data dependencies. However, the individual iterations are often independent so fine-grained parallelism should confer a significant performance gain. Compiler directives tend to break loops into separate UNIX processes with fork/join operations. These processes are usually linked to physical processors and require a significant amount of overhead to create and destroy.

Another model of execution is threads. A thread represents an instruction stream executing within a shared address space that can be used to accomplish a given task. Execution of multiple threads can yield efficient resource utilization and require less system overhead to maintain. One of the most common uses for threads is to overlap computation with I/O. In this case, a process creates one thread to perform I/O and another to continue with calculations. The threads are then executed concurrently. For example, when the I/O thread is waiting for the completion of input, the computation thread executes; when the computation thread waits for a memory fetch, the I/O thread executes.

Threads are sometimes called “lightweight” processes because they share many of the properties and attributes of full processes. When a processor switches context between two processes, the entire memory space of the executing process must be saved and the memory space of the process scheduled for execution must be restored. However, since multiple threads created by the same process share the memory address space of that process, there is no need to save and restore large portions of memory when switching context between threads. This savings of processor time and resources is one of the major advantages of programming with threads.

Pthreads is a POSIX standard established to control the spawning, execution and termination of multiple tasks within a single process (Nichols et al., 1996). Concurrent tasks are assigned to independent threads. Threads within the same process have access to their own local, private memory but also share the same memory space of the global process. Executing on shared-memory computers, the system may be able to assign the threads to run across multiple processors.

---

[1] CEWES MSRC On-site Parallel Tools Lead for PET, Rice University
[2] CEWES MSRC On-site CFD Lead for PET, Mississippi State University

As useful as the Pthreads standard is for parallel programming on shared-memory computers, a Fortran interface is not defined. However, there are no serious technical barriers to implementing such an API (Application Programming Interface). We describe the implementation and testing of Fortran 90 bindings for the POSIX threads library under development at the CEWES MSRC. Section 2 gives a brief overview of the POSIX threads programming model and the routines contained in the standard. Next, a description of how the Fortran 90 interface routines are implemented is given in Section 3. Some preliminary results of using the bindings within an actual particle dynamics code are given in Section 4. A comparison of these results with other parallelization methods on the same code will also be presented. Section 5 describes our preliminary investigation into combining MPI and Pthreads. Finally, our conclusions and directions for future research are presented in Section 6.

2. The POSIX Threads Library

2.1 The Pthreads Programming Model

Threads are used for task or function parallelism. A single process made up of many independent tasks may break up its computation into a set of concurrently executing threads. Each thread is created and assigned a given subroutine (parameters may be sent to the subroutine if needed); the subroutine code is executed concurrently with all other active threads, and the thread is deleted after completion. Each thread is an instruction stream, with its own stack, sharing a global memory space. Independent loop iterations (encapsulated within a subroutine call) can be executed as threads. This is the basis for our use of threads within the particle dynamics code.

All threads executing within a process are peers. There is no explicit parent-child relationship. A thread may be halted until another thread completes its task, but ultimately, all threaded tasks are concurrent. The operating system performs scheduling. However, Pthreads functions are available to set priorities and alter scheduling.

2.2 Library Details

The library is relatively small, consisting of only 61 routines that can loosely be classified into three categories: thread manipulation (e.g., creation, termination), synchronization (e.g., locks, barriers), and scheduling (e.g., execution priority). Each thread is given a unique thread ID upon creation. The Pthreads standard defines attributes in order to control the execution characteristics of threads. Such attributes include detach stack, stack address, stack size, scheduling policy and execution priorities.

3. Fortran 90 Bindings

The interface we propose consists of two files. The first is a Fortran 90 module containing some necessary constants and derived type definitions. The second is a collection of wrapper routines that provide the Fortran bindings to the Pthreads library. These wrappers are void C functions that call the vendor-supplied Pthreads library. The wrappers are called from Fortran 90 programs as if they were external subroutines.

Many software libraries have incorporated both a Fortran and C interface to the same functions regardless of the language in which the libraries are originally implemented. We have taken a cue from some of these previous efforts; for example, Sunderam et al. (1994) and Snir et al. (1997). The C wrapper functions are given the same names as the corresponding POSIX routines with an “f” prefixed. For example, a C thread obtains its thread ID with the call:

```
my_id = pthread_self();
```

whereas a Fortran thread gets its thread ID as follows:

```
call fpthread_self(myid).
```

Also, most C routines return an error code via the function name, thus an additional parameter is added to the end of the Fortran parameter list in order to return this error code.[3]

When designing routines written in one language, to be called from code written in another language, matching the methods utilized for the passing of parameters is an important consideration. Fortran 90 uses *pass by reference* for all its parameters while C may use *pass by value*. This difference requires that all C wrapper functions assume the parameters sent would be passed by reference; i.e., all dummy parameters are pointers. We were not tempted to mix Fortran 90 pointers with C pointers, although this may have simplified some of the wrapper routines. Reliance on a straightforward compatibility between the two languages would be risky and would likely reduce the portability of the bindings.

Another portability issue is how compiled names are denoted in the object code created by the compiler. The Fortran 90 compiler on the SGI Origin 2000 appends an underline character to the name of compiled functions. Thus, each of the C wrapper function names is suffixed with '_'. Other systems use all capital letters while others do not perform any alterations. To handle this difference between compilers, the wrapper source code shall contain C preprocessor commands that will be able to select the appropriate name for functions.

A further technical difficulty is that the Pthreads library makes extensive use of C structures in its definition of Pthreads data types. The Pthreads standard defines data types to handle such things as thread attributes, mutex and conditional variables, and mutex and conditional attributes. To preserve the Pthreads types defined within the bindings, a Fortran 90 derived type is defined with the same naming conventions used to name the wrapper routines; e.g., `pthread_mutex_t` becomes `fpthread_mutex_t`. Only the functions within the Pthreads library manipulate the data in these structures. The Fortran 90 application program rarely needs to access the data directly. Instead, it must pass the address of a data type to a library routine. Thus, rather than trying to pass Fortran 90 derived types composed of the appropriate components into C structures, the derived type contains only a simple integer, which stores an address. Within the Fortran 90 module, this integer is often declared to be **`PRIVATE`**. This adds an extra level of security to these derived types by preventing the programmer from inadvertently changing the address of an important variable.

The wrappers to the POSIX functions interpret this integer as the address of the C structure required. The wrapper code must, where appropriate, allocate space on the heap for the needed structure, decode the integer parameter to be a pointer to a structure, modify the contents of the structure and free up the space when the structure is no longer needed. Thus, only the structure locations are communicated between the Fortran 90 application and the interface. This improves the portability of the bindings and the application programmer need not worry about any differences between Fortran 90 and C pointers or derived types and structures.

An example of this information hiding is illustrated in the following two wrapper codes. The first is **`fpthread_mutex_init_`** which allocates memory to hold the mutex structure (`pthread_mutex_t`), initializes the structure by calling the Pthreads function, and returns the address (as an integer) of the initialized mutex to the calling Fortran 90 code.

```
void fpthread_mutex_init_(int *mutex, int *attr, int *ierr)
{
      pthread_mutex_t *lmutex;
      pthread_mutexattr_t *lattr;

      lmutex = (pthread_mutex_t *) malloc( sizeof (pthread_mutex_t));
      lattr = (pthread_mutexattr_t *) (*attr);
      *ierr = pthread_mutex_init(lmutex, lattr);
      *mutex = (int) (lmutex);
}
```

[3] `pthread_self` is the only Pthreads routine whose error code is the same as the desired return value; thus there is no extra parameter in the Fortran 90 binding.

The second wrapper is **`fpthread_mutex_destroy_`** which returns the memory allocated to the mutex back to the system and returns the NULL pointer to the calling Fortran 90 code.

```
void fpthread_mutex_destroy_(int *mutex, int *ierr)
{
      pthread_mutex_t *lmutex;

      lmutex = (pthread_mutex_t *) (*mutex);
      *ierr = pthread_mutex_destroy(lmutex);
      free(lmutex);
      *mutex = NULL;
}
```

The programmer must pay close attention to the scope of variables since the rules for scoping are different in Fortran and C. This is particularly important when using mutexes and conditional variables. In order to ensure that only a single thread is allowed to enter a critical section of the code, a single copy of a lock must be declared in memory and visible to all threads that make use of it. Such global variables are easily declared in C. With Fortran 90, it is suggested that all mutex locks and conditional variables be declared within a module that is used by each subroutine in which these synchronization constructs are needed.

4. Preliminary Results

The high-energy impact simulation program, MAGI, is used as a test bed for the Pthreads API. MAGI is a smoothed-particle dynamics hydrocode. Dynamics algorithms, in general, are difficult to parallelize. The through-space interactions between neighboring particles, which must be calculated during every time step, consume the bulk of computational time. However, the particle data tend to be unstructured and poorly aligned. To make matters worse, the nearest-neighbors change as the dynamics trajectory evolves. This makes data distribution and load balancing very difficult.

In MAGI, the through-space interactions are calculated within a single, large loop that contains numerous data dependencies and subroutine calls. The program spends most of its time here. Fortunately, each iteration of this loop is independent of all other iterations. Parallelizing this loop using Parallel Computing Forum (PCF) directives confers a significant speed-up. Table 1 shows timing results for a 100,000 particle simulation of a steel block striking an aluminum plate at high velocity. The same simulation was performed expressing loop-level parallelism using Pthreads (Table 2). The pre-processor performance is lower for threads, but the scalability appears better than for PCF directives. In this experiment, a create/join model is used to thread the particle loop. So, the system overhead associated with creating the threads is paid at every time step. It is possible that a more efficient thread pool model will improve the performance of the Pthreads version of MAGI. In a thread pool algorithm, the threads are created at the start of the simulation and reused.

Table 1. Timings for a MAGI simulation on a 16-CPU SGI Power Challenge (level-3 optimization and PCF directives).

# CPU	Time (seconds)
1	862
2	583
4	475
8	425
12	405
16	403

Table 2. Timings for a MAGI simulation on a 16-CPU SGI Power Challenge (level-3 optimization and Pthreads).

# Threads	Time (seconds)
1	1299
2	1031
4	725
8	557
16	473
24	402
32	445
48	511

5. Pthreads and MPI

A current trend in the future of HPC architectures is the cluster of symmetric multi-processor (SMP) nodes connected to each other via a network. Each SMP node is a shared memory multi-processor that can make use of Pthreads. In order to share data between nodes, some explicit message passing must be done. Thus, a combination of both Pthreads (local to SMP nodes) and MPI (message passing between nodes) is a potential model of computation for effectively programming on SMP cluster platforms.

In order to test the efficacy of such a hybrid model, we have developed small test codes that use the Fortran 90 Pthreads bindings and MPI calls. Our experiments were carried out on the SGI/Cray T3E at CEWES MSRC. Each node of the T3E simulated a SMP cluster running multiple threads while data was shared between nodes via MPI calls. Each thread was able to communicate directly with other threads on other processors. The rank of the processor was used to address messages toward a pool of threads executing on the processor. Tag numbers were then used by individual threads to receive messages that were specifically addressed to that thread.

These experiments were simply a proof-of-concept exercise. We are unaware of any previous experiments combining Pthreads and MPI calls, which gives us nothing to compare to or upon which to improve. Further development and testing of this model is necessary.

6. Conclusions

Pthreads is a convenient method of expressing task-level parallelism. The programmer has more control over scheduling and synchronization with Pthreads than with parallel compiler directives. More important, the programmer can control the grain size of each thread. If the grain is too coarse, more threads can be created and vice versa. Pthreads has the advantage of low system overhead because all threads exist in a single UNIX process. In addition, multiple threads can exist on a single processor, thus giving better resource utilization.

Threading is widely used in systems programming, where C/C++ is the primary language. However, Fortran is the primary language for high performance computing (HPC). Threading is not as common in HPC applications because the Posix standard does not define a Fortran interface to the Pthreads library. The Fortran API described here provides an additional parallel programming tool to the DoD HPC community. In addition, combining Pthreads with MPI presents an attractive programming model for SMP clusters.

Acknowledgements

This work was funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through Programming Environment and Training (PET), Contract Number: DAHC 94-96-C0002, Nichols Research Corporation. The authors wish to thank David Medina (USAF Research Lab) and Ted Carrney (Energetic Materials Research and Testing Center, New Mexico Tech) for assistance with the MAGI code.

References

Nichols, B., Buttlar, D. and Farrell, J., *Pthreads Programming*, O'Reilly and Associates, Inc., Sebastopol, CA, 1996.

Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W. and Dongarra, J., *MPI: The Complete Reference*, The MIT Press, Cambridge, Massachusetts, 1997.

Sunderam, V. S., Geist, G. A., Dongarra, J. and Manchek, R., "The PVM Concurrent Computing System: Evolution, Experiences, and Trends," *Parallel Computing*, Vol. 20, No. 4, pp.531-545, April 1994.